GPU-enabled Length-Aware Cuckoo Filter for Faster IP Lookup

Satyajeet Shahane
Advisor: Dr. Minseok Kwon

ABSTRACT

As network speeds continue to grow, forwarding table lookup proves to be slow and acts as a bottleneck for the overall throughput. Unlike physical routers, lookup in software routers cannot be performed entirely by hardware. The large size of the forwarding table mandates it to be stored off-chip. Many off-chip accesses for longest prefix lookup make this a very slow process. Earlier attempts to reduce off-chip memory access mainly revolve around bloom filters and their modifications. Bloom filters have a limitation that they do not permit dynamic deletion of entries. To overcome these, we propose a GPU based implementation of a Length Aware Cuckoo filter. LACF not only reduces the off-chip memory accesses but also find the longest prefix very quickly using the power to the GPU. Our results show that the GPU-enabled LACF is a scalable technique that has lessers number of false positives than other lookup techniques, is more time efficient than them and also uses less hash probes.

INTRODUCTION

To find the right destination port for an incoming packet, the router needs to find the longest prefix (of IP of the packet) in the forwarding table. Forwarding tables store a mapping of IP prefixes to forwarding interface and are typically very huge in size. Without any filtering technique, longest prefix matching would entail a worst case of 32 off-chip accesses. These high number of off-chip memory accesses are the primary cause of the delay in fast packet processing.

To reduce the number of off-chip memory accesses, previous research works have mainly focused on using Bloom filters. Bloom filter is a probabilistic set membership data structure. Bloom filters also have a very small memory footprint. The idea is to construct a bloom filter from all entries in forwarding table and perform membership check for all prefixes and query the actually forwarding table only for the longest prefix which was found to be a member. However if there are a high number of false positives this increases the number of off-chip memory access. Also, for a standard bloom filter dynamic removal of entries is not possible without rebuilding the entire filter. There are variants of the standard Bloom filter e.g.,
Counting bloom filter, d-left counting bloom filters, quotient filters which support dynamic removal of entries but at a much higher space cost.

Cuckoo filter uses cuckoo hashing which can be used to perform membership tests with lower false positive rates and with only a marginal increase in the space cost. Reduced false positive rate reduces the number off-chip memory access thus increasing the efficiency and processing packets faster. To reduce the time spent in performing filter lookup, it is implemented on a GPU. This enables parallel lookup of all 32 prefixes, greatly speeding up the overall lookup process. The remaining paper is organized as follows - Section 2 discusses the related work and background on Cuckoo hashing. Section 3 discusses the cuckoo filter creation and lookup algorithm. Section 4 discusses the detailed implementation using a GPU. Section 5 discusses the results which is followed by sections 6 and 7 giving the future work and conclusions that can be drawn.

BACKGROUND

This section also gives some background on cuckoo hashing and basic GPU architecture which are essential to understand the remaining contents of this paper.

1. **Cuckoo hashing**: The purpose of hashing is to map a key to a bucket using a hash function. Cuckoo hashing can involve \(\text{n}\) number of hash function. Let us assume \(\text{n} = 2\) for simplicity sake. The two hash functions give two possible buckets for key. If either of the two buckets are empty the key is placed there. If not then one location is chosen and the key is placed there. The existing key gets kicked out and is placed at its alternate location, provided it is free. If not the same procedure is repeated. If a certain predefined number of max kicks is exceeded then the table is declared as full.

![Figure 1](image)

**Figure 1**: Cuckoo hashing: kicks out an element to accommodate another [2]
2. **Basics of GPU architecture**: A Graphics Processing Unit has several multiprocessor inside it. These multiprocessors are connected to a global memory. The CPU interacts with the GPU using drivers like CUDA and places any data it wants to give to the GPU in this global memory. Inside every multiprocessor, there are several cores. These cores share a very fast memory and also have a register bank.

![Diagram of GPU architecture]

*Figure 2: General architecture of a GPU [3]*

**LACF Algorithm**

The length aware cuckoo filter algorithm has two main parts, filter creation and filter lookup. We look at both of them in detail.

1. **Filter creation**: The filter is created from the prefix entries in forwarding table. LACF uses partial key cuckoo hashing. Using this technique, the alternate location in filter can be calculated using just the hash of the fingerprint and the first location [2]. Insertions are done based on popularity of IP prefix. Unpopular prefixes are inserted twice and popular only once. This reduces the false positive rate for unpopular IP prefixes without much increase in the false positive rate of popular prefixes.
2. **Filter lookup**: Lookup for an unpopular IP requires us to check both the bucket locations for that IP. Popular IP is searched only in either of the two. Searching the filter for all 32 prefixes is done in parallel using a GPU. The GPU then returns a small list of the possible longest prefixes which are checked in the actual hash table.

---

**Algorithm 1**: Algorithm to insert in LACF

\[
\begin{align*}
f &= \text{fingerprint}(x) \\
i1 &= \text{hash}(x) \\
i2 &= i1 \oplus \text{hash}(f)
\end{align*}
\]

\[
\text{if \ bucket[i1] or bucket[i2] has an empty entry then} \\
\quad \text{add } f \text{ to that bucket} \\
\quad \text{return Done}
\]

//need to relocate existing filter entry i = randomly pick i1 or i2;
for \( n = 0; n < \text{MaxNumKicks}; n++ \) do
\quad \text{randomly select an entry } e \text{ from bucket[i]}
\quad \text{swap } f \text{ and the fingerprint stored in entry } e
\quad i = i \oplus \text{hash}(f)
\quad \text{if } \text{bucket[i] has an empty entry then}
\quad \quad \text{add } f \text{ to bucket[i]; return Done;}
\end{align*}
\]

\[
\text{if un-popular ip prefix then} \\
\quad \text{insert into previously un-inserted i1/i2}
\]

\[
\text{if MaxNumKicks exceeded then} \\
\quad \text{hashtable declared to be full}
\]

---

**Algorithm 2**: Algorithm for LACF parallel lookup

\[
\begin{align*}
f &= \text{fingerprint}(x) \\
i1 &= \text{hash}(x) \\
i2 &= i1 \oplus \text{hash}(f)
\end{align*}
\]

\[
\text{if bucket[i1] or bucket[i2] has } f \text{ then} \\
\quad \text{return True}
\]

\[
\text{return False}
\]
IMPLEMENTATION

During the lookup operation, cuckoo filter is searched for all possible 32 prefixes. The idea is to start searching the cuckoo filter with the longest prefix (i.e., 32) and progress towards the smallest prefix (i.e., 1). Every time the filter returns true for a particular prefix, that prefix is stored in a list. Once the filter is searched for all 32 prefix lengths, the list contains all those prefixes (usually two to four) for which the filter returned true. Now the actual forwarding table is searched only for the prefix lengths stored in this list.

Performance of LACF is improved further by querying the filter for all 32 prefixes in parallel. A GPU is used to achieve this parallel lookup. The GPU consists of a set of multiprocessors and a global memory shared by them. Each multiprocessor in turn consists of several cores. The multiprocessor also contains a very fast shared memory which is accessible only to the cores within that multiprocessor. The filter resides in every multiprocessors shared memory while the actual forwarding table is in the CPU main memory.

The code for LACF is organized into two modules. The kernel module is written in C which runs in every core of every multiprocessor in the GPU. This module takes an IP address as input and performs parallel lookup from the filter stored in the shared memory. The driver module, written in java, invokes the kernel module using parallel java 2. The kernel module returns a list of possible prefix lengths to the driver module (executed in the CPU), which then looks up the actually forwarding table which is stored in the main memory. The fact that the CPU main memory lookup is reduced from a worst case of 32 to a worst case of just 2 or 3 lookups is the main reason for speedup. The code for filter creation is also present in the driver module.

![Figure 3: System implementation for parallel IP lookup](image)
**SETUP ENVIRONMENT**

Tesla C2075 is the GPU used. It has 14 multiprocessors, each connected to a global memory, 6GB in size. The global memory is connected to the CPU memory using a high speed bus. Each multiprocessor has 32 cores; a register bank with 32,768 high-speed 32-bit registers; a shared memory; and an L1 cache. The shared memory and the L1 cache together total to 64KB. We configure the multiprocessors to keep entire 64KB for shared memory. This is done in-order to allow maximum space for the filter in the shared memory. The programs are executed on x86 Ubuntu machines having 3.13.0-100 kernel version. The Java Runtime Environment version used for performing all experiments is seven.

**RESULTS AND PERFORMANCE EVALUATION**

Performance of LACF is measured using lookup time and hash probes as metrics. Lookup time is the time needed to lookup a set of IP addresses from the forwarding table, using the filter. Hash probe measures the average number of times the forwarding table is queried for one IP address. The performance of different lookup techniques have been evaluated on a data set which contains entries obtained from actual router’s forwarding tables. Four forwarding tables having 5,000, 10,000, 15,000 and 20,000 number of entries respectively were used. For each of these forwarding tables, a lookup for 10,000 IP addresses was performed. To get an accurate value of the time taken by the different lookup techniques, the average of 50 runs is taken. For BF and hashing, lookup was done for 1000 entries and the result extrapolated (to avoid unnecessary optimization enforced by JVM).

The time taken by LACF and CF lie in the range of 150ms to 160ms and 170ms to 175ms respectively, while the time taken for BF and hashing lie in the range of 170ms to 175ms and 170ms to 180ms respectively. The average hash probe values per IP lookup for LACF and CF lie in the range of 1.1 to 1.6 and 1.7 to 3.7 respectively, while the values for BF and hashing lie in the range of 6.2 to 8.5 and 10.5 to 11 respectively. The values of standard deviation also are computed for each algorithm during the 50 test runs. The standard deviation values in milliseconds for LACF and CF lie in the range of 1.6 to 2.7 and 7.1 to 8.7 respectively, while the values for BF and hashing lie in the range of 8.7 to 9 and 3 to 5 respectively.

It can be inferred from these results that LACF performs better than other lookup techniques. The reason for this is that, of the total time taken by LACF, majority of it is spent in filter lookup. This reduces the number hash probes, which is a costly operation. The comparison of times spent in filter lookup and hash table lookup are seen in Figure 3.

A further detailed investigation of the time taken for LACF program execution revealed that majority of the was spent in memory I/O, that is data transfer from CPU to GPU and vice
versa. On an average, for every lookup operation, about 45\% of the total time is spent in memory I/O.

**Figure 4**: Average number of hash probes Versus number of entries in the forwarding table. This is measured for a lookup for 10,000 IP addresses.
Figure 5: Time (in milliseconds) Versus number of entries in the forwarding table. This is measured for a lookup for 10,000 IP addresses.

Figure 6: Filter-Hash table time split versus Lookup technique. This is measured for a lookup of 10,000 IP addresses in each case.

<table>
<thead>
<tr>
<th>Lookup technique</th>
<th>False positive rate</th>
<th>Average bits per item</th>
</tr>
</thead>
<tbody>
<tr>
<td>LACF</td>
<td>0.281%</td>
<td>7.5</td>
</tr>
<tr>
<td>CF</td>
<td>0.9%</td>
<td>5</td>
</tr>
<tr>
<td>BF</td>
<td>0.944%</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 1: Space efficiency and false positive rate
CONCLUSION

Using a GPU-enabled version of a length aware cuckoo filter gives significantly better performance than GPU-enabled cuckoo filter or bloom filter. LACF beats the other techniques in time and number of hash probes. LACF has a better filter occupancy as well. In-spite of higher filter occupancy, LACF has a lower false positive rate when compared to the other lookup techniques.

The lookup time is mainly divided into two parts - time to lookup filter and time to lookup hash table which is stored off-chip. LACF spends more time in filter lookup as compared to others. However this enables to reduces number of hash probes, greatly reducing time spent is off-chip accesses. Thus the overall lookup time for LACF is lower than others. Time spent by LACF on the filter is reduced even more by doing 32 lookups in parallel. For every lookup done in the GPU, the CPU needs to transfer some data to the GPU global memory and then do the actual lookup. It was observed that 45% of the time needed for one single is spent in the data I/O from CPU to GPU. If this time is to be discounted, then the performance of parallel LACF would clearly be seen to be even better.

ACKNOWLEDGEMENT

I would like to thank Dr. Minseok Kwon for his invaluable inputs, guidance and motivation throughout the course of the project. I would also like to thank Prof. Alan Kaminsky for allowing me to use his GPU for experiment purposes and for providing guidance from time to time regarding parallel programming with CUDA.

REFERENCES

